Clase 6 - Recursión Optimizada - Programación dinámica

---

1. *Recursión de Cola*

Recursividad pero Optimizada!!! Recursión de Cola

repaso:

La **recursión** es una técnica ampliamente utilizada en el diseño de algoritmos donde una función se llama a sí misma, descomponiendo un problema grande en problemas más pequeños y manejables hasta que se alcanza una solución directa, conocida como caso base. La recursión es utilizada en problemas donde una estructura repetitiva natural está presente, como el cálculo del factorial de un número, la sucesión de Fibonacci, o la resolución del problema de las torres de Hanói.

Un algoritmo recursivo consta de:

- **Caso Base**: La condición que detiene la recursión. Evita que el algoritmo entre en un bucle infinito.
- **Caso Recursivo**: La parte del algoritmo donde se realiza la llamada a sí mismo, descomponiendo el problema en subproblemas más pequeños.

¿Qué es la Recursión de Cola?

La **recursión de cola** es una forma especial de recursión en la que la llamada recursiva es la **última operación** que se ejecuta en la función antes de retornar el valor al programa principal. En otras palabras, en una recursión de cola, no se realiza ningún cálculo adicional después de la llamada recursiva, lo que permite que el resultado de esta llamada sea el valor final.

El autor Thomas Cormen, en su libro *Introduction to Algorithms* (2009), destaca que una función recursiva de cola "es aquella en la que no quedan operaciones pendientes cuando se realiza la llamada recursiva. Esta propiedad permite optimizaciones de memoria que no son posibles en una recursión estándar". Por lo tanto, cuando un algoritmo es implementado como una recursión de cola, muchos compiladores y lenguajes pueden optimizar la ejecución del programa, transformando la recursión en un bucle interno.

¿Por qué es una optimización?

La recursión de cola es una optimización importante porque permite mejorar el **rendimiento en tiempo y memoria** de los algoritmos recursivos. Durante la ejecución de una función recursiva estándar, cada llamada recursiva genera un nuevo marco en la pila de ejecución, donde se almacenan las variables locales y el estado de la función. Si una función recursiva se llama a sí misma demasiadas veces, se puede llenar la pila, causando un error de desbordamiento (StackOverflow).

Sin embargo, en la recursión de cola, no es necesario mantener múltiples marcos de pila, ya que no hay operaciones pendientes que deban ejecutarse después de la llamada recursiva. Esto permite que los lenguajes de programación, que soportan **optimización de recursión de cola (Tail Call Optimization, TCO)**, reemplacen las llamadas recursivas por un bucle iterativo, reutilizando el mismo marco de pila para cada llamada y, por lo tanto, reduciendo significativamente el uso de memoria.

La recursión de cola, en lenguajes que soportan esta optimización, puede manejar una gran cantidad de recursiones sin riesgo de desbordar la pila, ya que el compilador transforma la recursión en un bucle iterativo. Sin embargo, es importante mencionar que **no todos los lenguajes soporta esta implementación nativa,** lo que significa que las llamadas recursivas en, incluso si son de cola, pueden causar problemas de desbordamiento de pila si no se manejan adecuadamente.

Diferencias con la Recursión Tradicional

En una función recursiva tradicional, después de la llamada recursiva, es posible que haya más operaciones por realizar. Por ejemplo, en el cálculo del factorial de un número, la función espera el retorno de la llamada recursiva para multiplicar ese valor por el número actual:

```
1  def factorial(n):
2      if n == 0:
3          return 1
4      else:
5          return n * factorial(n - 1)  # Se espera que regrese el valor para multiplicarlo por n
```

En este ejemplo, **la multiplicación por n ocurre después de que la llamada recursiva retorna**, lo que significa que el marco de pila debe mantenerse hasta que la multiplicación se complete. Esto consume más memoria.

En contraste, en la **recursión de cola**, no hay cálculos pendientes después de la llamada recursiva, como se muestra a continuación:

```
7  def factorial_tail(n, acc=1):
8      if n == 0:
9          return acc
10     else:
11         return factorial_tail(n - 1, acc * n)  # El cálculo se realiza antes de la llamada recursiva
```

Aquí, la llamada a factorial_tail es la última operación que se ejecuta, lo que permitiría al compilador (si soportara TCO) optimizar el uso de la memoria.

Actividad Práctica

1. **Ejercicio 1**: Implementar una función recursiva de cola para calcular la suma de los primeros n números naturales.

Actividad de Investigación

Investiga si **Python** soporta la optimización de recursión de cola (TCO). Para ello, responde las siguientes preguntas:

¿Python incluye optimización de recursión de cola en su implementación estándar?
¿Existen herramientas o técnicas en Python que puedan ayudar a optimizar este tipo de recursión? (Por ejemplo, modificadores de la pila o uso de módulos externos).

2. Programación Dinámica - Memoización y Tabulación

Programación Dinámica | Memoización | Explicado Paso a Paso

Introducción

En la clase anterior se abordó el cálculo de la secuencia de **Fibonacci** utilizando la recursión, y observamos que esta técnica, aunque elegante, no es eficiente para problemas donde el mismo subproblema se resuelve varias veces, como ocurre en Fibonacci. En esta clase, estudiaremos dos técnicas fundamentales para optimizar problemas recursivos: la **memoización** y la **tabulación**, que forman parte de un paradigma llamado **programación dinámica**.

La programación dinámica es un enfoque que descompone un problema complejo en subproblemas más simples, resolviendo cada subproblema solo una vez y almacenando su solución para evitar cálculos repetidos. Según Cormen et al. (*Introduction to Algorithms*, 2009), "la programación dinámica es útil cuando el problema puede dividirse en subproblemas que se solapan, y la solución a cada subproblema contribuye a la solución del problema original".

Memoización

Definición: La **memoización** es una técnica de optimización que almacena el resultado de las llamadas recursivas en una estructura de datos (normalmente un diccionario o una tabla), de forma que si un subproblema ya ha sido resuelto, se puede recuperar su valor directamente sin tener que volver a calcularlo. Este enfoque se utiliza comúnmente en problemas donde el mismo subproblema se repite varias veces, como en Fibonacci.

En palabras de Cormen, "la memoización consiste en recordar las soluciones ya calculadas para los subproblemas, evitando que se resuelvan de nuevo, lo que optimiza enormemente el rendimiento en algoritmos recursivos con solapamiento de subproblemas".

Ejemplo en Python (Fibonacci con memoización): Para entender cómo funciona, implementaremos el cálculo de Fibonacci utilizando memoización:

```
15  def fibonacci_memo(n, memo={}):
16      if n in memo:
17          return memo[n]
18      if n <= 1:
19          return n
20      memo[n] = fibonacci_memo(n - 1, memo) + fibonacci_memo(n - 2, memo)
21      return memo[n]
```

En este ejemplo, el diccionario `memo` almacena los resultados de los cálculos de Fibonacci para evitar cálculos repetidos. Si se llama a `fibonacci_memo` para un valor que ya ha sido calculado, simplemente se devuelve el valor almacenado.

Ventajas de la Memoización:

- **Eficiencia temporal**: Reduce drásticamente la cantidad de llamadas recursivas, pasando de un algoritmo exponencial O(2^n) a uno lineal O(n), ya que cada subproblema se resuelve solo una vez.

- **Reutilización**: Los resultados de los subproblemas se almacenan y se reutilizan cuando sea necesario.

Cómo mejora el cálculo de Fibonacci: En la implementación recursiva básica de Fibonacci, cada llamada a `fibonacci(n)` desencadena dos llamadas más: `fibonacci(n-1)` y `fibonacci(n-2)`. Esto genera una duplicación exponencial de subproblemas. Sin embargo, con memoización, los valores previamente calculados se almacenan y reutilizan, lo que ahorra un enorme esfuerzo computacional.

Según Luis Joyanes Aguilar (*Fundamentos de Programación*, 2011), la memoización "es una técnica eficiente que permite reducir el costo computacional de los algoritmos recursivos, evitando el recalculo innecesario de soluciones intermedias ya conocidas".

Tabulación

Definición: La **tabulación** es otra técnica de programación dinámica que se basa en construir una tabla (generalmente un arreglo o una matriz) de abajo hacia arriba. A diferencia de la memoización, que es recursiva y resuelve el problema cuando es necesario, la tabulación comienza resolviendo todos los subproblemas más pequeños primero y almacena sus resultados en una tabla para usarlos en el cálculo de subproblemas más grandes.
Cormen señala que "en la tabulación, resolvemos todos los subproblemas pequeños antes de abordar los más grandes, evitando la recursión y garantizando que cada subproblema se resuelve una única vez".

Ejemplo en Python (Fibonacci con tabulación): A continuación, se muestra cómo implementar el algoritmo de Fibonacci usando tabulación:

```
23  def fibonacci_tabulation(n):
24      if n <= 1:
25          return n
26      table = [0] * (n + 1)
27      table[1] = 1
28      for i in range(2, n + 1):
29          table[i] = table[i - 1] + table[i - 2]
30      return table[n]
```

En este caso, el algoritmo construye una tabla `table` que almacena las soluciones de Fibonacci desde `fibonacci(0)` hasta `fibonacci(n)`. Cada valor se calcula solo una vez, lo que optimiza la ejecución en términos de tiempo y evita la sobrecarga de la pila que ocurre con la recursión.

Ventajas de la Tabulación:

- **Eficiencia espacial**: Aunque consume espacio proporcional al tamaño del problema, evita la sobrecarga de llamadas recursivas, lo que puede ser beneficioso en sistemas con recursos limitados.
- **Eliminación de la recursión**: A diferencia de la memoización, la tabulación no utiliza recursión, lo que elimina la posibilidad de desbordamiento de la pila en problemas de gran tamaño.

La tabulación es especialmente útil en entornos donde los problemas recursivos podrían llevar al desbordamiento de la pila, dado que evita la recursión, utilizando una aproximación iterativa.